

Lab 11: Topic: Analyzing C#Console Games for Bugs

Introduction, Environment, and Tools

Overview

The development of console-based games using C# provides a foundational understanding of programming logic, control flow, and software debugging. In this lab, we explored the structure and behavior of C# console game applications, leveraging Visual Studio's powerful debugging tools to examine the internal operations of game code. Debugging is a critical skill in software development, allowing developers to identify, isolate, and fix bugs that may otherwise cause crashes or incorrect behavior in a program. This lab focused on visualizing program execution, understanding how code flows during runtime, and learning how to detect and resolve issues in real-world codebases. By using open-source games from the official .NET console games repository, we gained hands-on experience with practical debugging strategies and problem-solving techniques.

Objectives

The main objective of this lab is to analyze C# console game applications and understand the benefits of using the Visual Studio Debugger. Specifically, the lab aims to:

- Explore how control flow operates in a C# console application.
- Demonstrate how to set breakpoints and navigate through code using the debugger (step-in, step-over, step-out).
- Identify and resolve bugs that lead to runtime crashes or logical errors.
- Reinforce debugging skills that are crucial in game development and general software engineering.

Environment Setup

To complete this lab, the following system and software configurations were used:

- **Operating System:** Windows 11
- **Development Environment:** Visual Studio 2022 (Community Edition)
- **.NET SDK:** Latest stable version compatible with Visual Studio 2022
- **Programming Language:** C# (latest stable version)
- **Project Repository:** <https://github.com/dotnet/dotnet-console-games> — A collection of console-based games built using C# and .NET.

Methodology and Execution

Bug Hunting and Fixing

Bug 1:

- **Game: Guess the number**
- **Bug Description:-**

Although the game specifies that the valid input range is between 1 and 100, it incorrectly accepts numbers outside this range (e.g., values less than 1 or greater than 100) as valid inputs. These invalid entries are still processed by the game, which could lead to an inconsistent or confusing user experience.

```
Guess a number (1-100): -100
Incorrect. Too Low.
Guess a number (1-100): 10000
Incorrect. Too High.
Guess a number (1-100): 0
Incorrect. Too Low.
Guess a number (1-100): |
```

- **Cause:** The source code does not include a condition to validate user input before it is compared to the randomly generated target number. As a result, out-of-bound numbers are accepted without any warnings or corrections.

```
Console.Write("Guess a number (1-100): ");
bool valid = int.TryParse((Console.ReadLine() ?? "").Trim(), out int input);
if (!valid) Console.WriteLine("Invalid.");
else if (input == value) break;
else Console.WriteLine($"Incorrect. Too {(input < value ? "Low" : "High")}");
```

- **Analysis and Fix:** To resolve this issue, a simple validation check was introduced immediately after the user inputs a number. This condition ensures that only numbers within the specified range (1–100) are accepted. If the input falls outside this range, the program prompts the user to enter a valid number, effectively preventing the game logic from processing invalid inputs.

```
Console.Write("Guess a number (1-100): ");
bool valid = int.TryParse((Console.ReadLine() ?? "").Trim(), out int input);
if (!valid || input < 1 || input > 100) Console.WriteLine("Invalid.");
else if (input == value) break;
else Console.WriteLine($"Incorrect. Too {(input < value ? "Low" : "High")}");
```

```
D:\Courses\STT\dotnet-consoc x + v
Guess a number (1-100): -100
Invalid.
Guess a number (1-100): 0
Invalid.
Guess a number (1-100): 10
Incorrect. Too Low.
Guess a number (1-100): 20000
Invalid.
Guess a number (1-100): |
```

Bug 2: Game Continuation Despite Repeated Key Press in "Clicker" Game

- **Game:** *Clicker*
- **Bug Description:**

According to the game instructions, the game should terminate if the same key is pressed consecutively. However, in its current state, the game continues running even when the same key is pressed multiple times in a row.
- **Cause:**

The source code lacks a conditional check to compare the current key press with the previous one. As a result, repeated key presses do not trigger any end-game behavior, which contradicts the intended game rules.

```
Console.CursorVisible = false;
ConsoleKey key = Console.ReadKey(true).Key;
switch (key)
{
    case >= ConsoleKey.A and <= ConsoleKey.Z:
        int index = Array.IndexOf(keys, (char)key);
        if (index < keyCount && key != previous)
        {
            previous = key;
            clicks += index > 1 ? BigInteger.Pow(10, index - 1) / (index - 1) : 1;
        }
        break;
    case ConsoleKey.Escape: goto MainMenu;
    default: goto ClickerInput;
}
```

- **Analysis and Fix:**

The issue was resolved by introducing a condition to check whether the current key is the same as the previous key (`key == previous`). If this condition is met, the game is terminated as expected. After implementing this logic, the game now correctly ends when a key is pressed twice in succession, aligning with the intended design.

```
ClickerInput:
Console.CursorVisible = false;
ConsoleKey key = Console.ReadKey(true).Key;
switch (key)
{
    case >= ConsoleKey.A and <= ConsoleKey.Z:

        if (key == previous)
        {
            TimeSpan duration = DateTime.Now - start;
            Console.Clear();
            Console.WriteLine(
                $""
                Game over! You pressed [{key}] twice.
                Final score: {clicks}
                Time Played: {duration}
                "");
            GameOver:
            switch (Console.ReadKey(true).Key)
            {
                case ConsoleKey.Escape: goto MainMenu;
                default: goto GameOver;
            }
        }
    int index = Array.IndexOf(keys, (char)key);
```

Game over! You pressed [Q] twice.

Final score: 3

Time Played: 00:00:13.3132446

Bug 3: Bullet Collision Issue in "Tanks" Game

- **Game:** Tanks
- **Bug Description:**

In certain situations, bullets are able to pass through walls when the tank is positioned directly adjacent to them. This behavior is unintended and breaks the game's logic, as bullets should not be able to penetrate solid obstacles like walls.



- **Cause:-**

The current implementation does not include a proper collision check between the bullet and the wall at the moment the bullet is fired. As a result, when the tank is placed right next to a wall, the bullet may be spawned on the opposite side of the wall, bypassing any collision logic altogether.

```

foreach (var tank in Tanks)
{
    #region Shooting Update

    if (tank.IsShooting)
    {
        tank.Bullet = new Bullet()
        {
            X = tank.Direction switch
            {
                Direction.Left => tank.X - 3,
                Direction.Right => tank.X + 3,
                _ => tank.X,
            },
            Y = tank.Direction switch
            {
                Direction.Up => tank.Y - 2,
                Direction.Down => tank.Y + 2,
                _ => tank.Y,
            },
            Direction = tank.Direction,
        };
        tank.IsShooting = false;
        continue;
    }

    #endregion
}

```

- **Analysis and Fix:**

To correct this issue, a collision check must be added before spawning a bullet. The game should verify whether the path is blocked by a wall at the firing position. If a wall is present, the bullet should not be spawned. Only when the path is clear should the bullet be allowed to appear and continue its movement. This fix ensures bullets behave realistically and respect environmental boundaries.

```

if (tank.IsShooting)
{
    int spawnx = tank.Direction switch
    {
        Direction.Left => tank.X - 3,
        Direction.Right => tank.X + 3,
        _ => tank.X,
    };

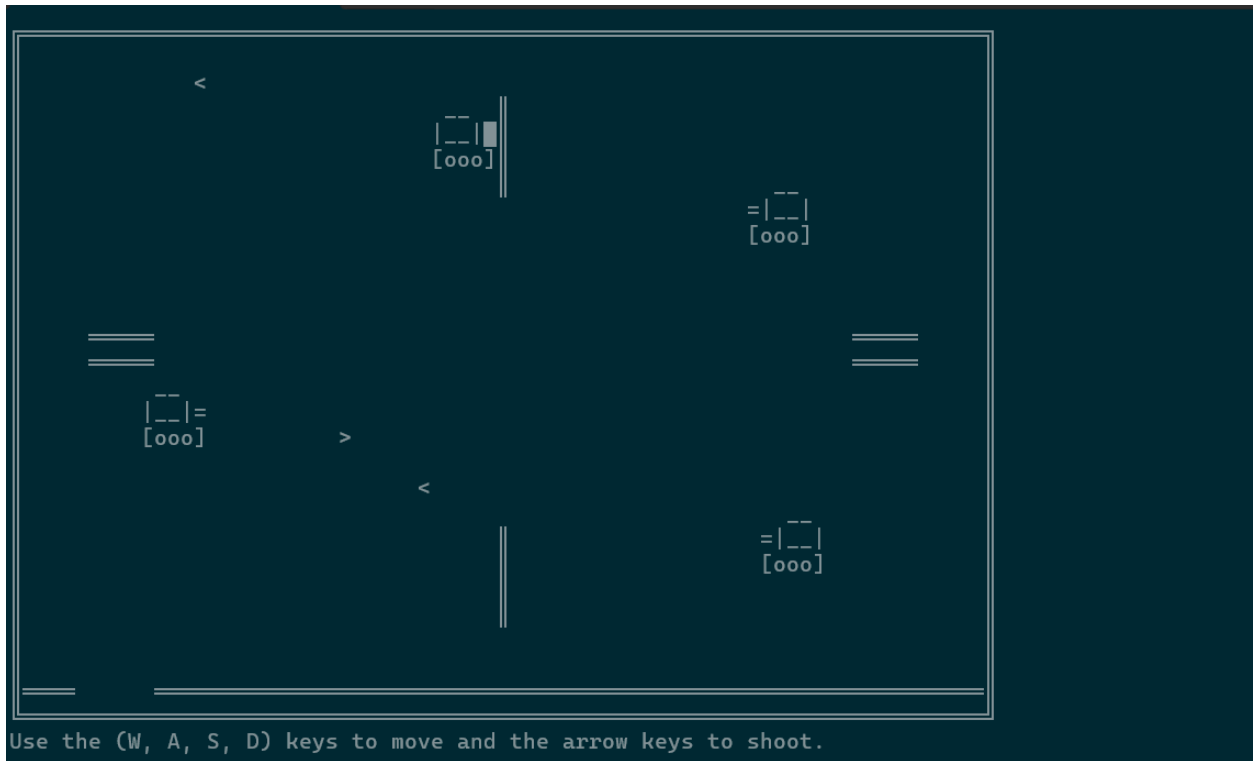
    int spawny = tank.Direction switch
    {
        Direction.Up => tank.Y - 2,
        Direction.Down => tank.Y + 2,
        _ => tank.Y,
    };

    bool block =
        spawnx <= 0 || spawnx >= 74 || spawny <= 0 || spawny >= 27 ||
        (5 < spawnx && spawnx < 11 && spawny == 13) ||
        (spawnx == 37 && 3 < spawny && spawny < 7) ||
        (spawnx == 37 && 20 < spawny && spawny < 24) ||
        (63 < spawnx && spawnx < 69 && spawny == 13);

    if (!block)
    {
        tank.Bullet = new Bullet()
        {
            X = spawnx,
            Y = spawny,
            Direction = tank.Direction
        };

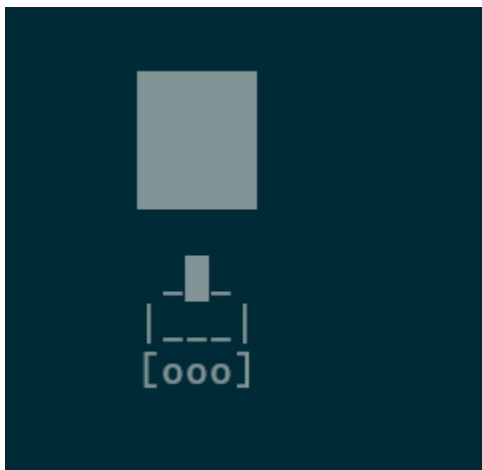
        tank.IsShooting = false;
        continue;
    }
}

```

Bug 4: Dead Tank Remains Visible After Health Depletion in "Tanks" Game

- **Game:** Tank
- **Bug Description:**
When a tank's health reaches zero, it should begin its explosion animation and then disappear. However, in the current implementation, the tank remains visible indefinitely, giving the impression that it never fully "dies."



- **Cause:**

The issue arises due to an incomplete condition check before setting the `collisionTank.ExplodingFrame` value. The code continues to reset `ExplodingFrame` to 1 every frame, even if it's already been set, which prevents the explosion animation from progressing and completing as intended.

```

        : Bullet[(int)bullet.Direction]);
    if (collision)
    {
        if (collisionTank is not null && --collisionTank.Health <= 0)
        {
            collisionTank.ExplodingFrame = 1;
        }
        tank.Bullet = null;
    }
}

```

- **Analysis and Fix:**

This bug was resolved by adding a simple condition to check if `ExplodingFrame` is equal to 0 before setting it to 1. This ensures that the explosion sequence only starts once and is allowed to proceed naturally. Preventing the frame from being reset in every cycle allows the tank to complete its destruction animation and properly disappear from the game screen.

```

        : Bullet[(int)bullet.Direction]);
    if (collision)
    {
        if (collisionTank is not null && --collisionTank.Health <= 0 && collisionTank.ExplodingFrame == 0)
        {
            collisionTank.ExplodingFrame = 1;
        }
        tank.Bullet = null;
    }
}

```

Bug 5:-

- **Game:** Helicopter
- **Bug Description:**

When drawing explosions or clearing old frames, the `Erase(string)` routine sometimes does `Console.SetCursorPosition(x, ++y)` with `y` equal to the console's bottom buffer line, causing an unhandled *ArgumentOutOfRangeException: The value must be ... less than the console's buffer size.*



Helicopter shape inside valid console buffer size



When it tries to exceed out-of-bound values

- **Cause:**

The code unconditionally increments y on every newline in the sprite string without verifying that $y + 1$ is still within *Console.BufferHeight*, so at buffer bottom it tries to position the cursor outside valid bounds.

```
void Erase(string @string)
{
    int x = Console.CursorLeft;
    int y = Console.CursorTop;
    foreach (char c in @string)
        if (c is '\n')
            Console.SetCursorPosition(x, ++y);
        else if (Console.CursorLeft < width - 1 && c is not ' ')
            Console.Write(' ');
        else if (Console.CursorLeft < width - 1 && Console.CursorTop < height - 1)
            Console.SetCursorPosition(Console.CursorLeft + 1, Console.CursorTop);
}
```

- **Analysis and Fix:-** By guarding the newline branch with a check against `Console.BufferHeight`, we prevent any attempt to move past the last buffer row. If `y` is already at `BufferHeight - 1`, we simply stop erasing further lines. This change eliminates the exception and restores smooth rendering.

```
void Erase(string @string)
{
    int x = Console.CursorLeft;
    int y = Console.CursorTop;
    foreach (char c in @string)
        if (c is '\n')
        {
            // Only move down if still within buffer height
            if (y < Console.BufferHeight - 1)
                Console.SetCursorPosition(x, ++y);
            else
                break; // stop to avoid ArgumentOutOfRangeException
        }
        else if (Console.CursorLeft < width - 1 && c is not ' ')
            Console.Write(' ');
        else if (Console.CursorLeft < width - 1 && Console.CursorTop < height - 1)
            Console.SetCursorPosition(Console.CursorLeft + 1, Console.CursorTop);
}
```

Result and Analysis

During this lab, five critical bugs were identified across different C# console games. Each bug was carefully analyzed using Visual Studio's debugging tools, and appropriate fixes were implemented. The bugs ranged from invalid input handling and logic omissions to faulty game behavior due to missing condition checks. After fixing each bug, the corresponding games were rebuilt and tested to verify that the issues were resolved. Using breakpoints, step-in/step-over/step-out operations, and runtime observation proved essential in pinpointing the exact causes of unexpected behaviors.

The corrected issues resulted in significant improvements in-game behavior, including:

- Proper validation of user inputs.
- Accurate collision detection.
- Correct end-game triggering.
- Logical handling of object states (like tanks disappearing after an explosion).

Discussion and Conclusion

This lab emphasized the importance of robust code logic and the power of debugging tools in game development. Even simple games can behave unpredictably when logic gaps are overlooked, especially in real-time or event-driven systems. By stepping through the code using Visual Studio Debugger, we were able to observe how the program executed line-by-line and where it deviated from expected behavior.

The hands-on process of identifying and fixing bugs reinforced that:

- Small oversights (like a missing condition) can cause major logic errors.
- Debugging tools are indispensable for effective software development.
- Understanding code flow is just as important as writing functional code.

Overall, this lab not only helped in refining debugging skills but also highlighted the real-world challenges developers face during game development.

Learnings

From this lab, the following key learnings were gained:

- **Practical Debugging:** Gained experience with Visual Studio Debugger operations such as breakpoints, step-in, step-over, and step-out.
- **Bug Analysis:** Learned how to read and understand source code in order to identify logical errors and unexpected behavior.
- **Code Correction:** Practiced implementing precise code fixes and verifying their effects through re-execution and testing.
- **Game Logic Insight:** Understood how fundamental programming concepts like condition checking, loops, and state management affect gameplay mechanics.

Summary

This lab provided valuable insight into debugging C# console games using Visual Studio. We could apply theoretical knowledge in a practical setting by working through real bugs in publicly available games. The experience improved both our coding and debugging skills, reinforced the importance of logical thinking in game development, and showcased how even minor issues can impact the user experience. We ensured each game performed as intended through systematic analysis and step-by-step correction, ultimately strengthening our capabilities as developers.

Lab 12: Event-driven Programming for Windows Forms Apps in C#

Introduction, Environment, and Tools

Overview

Event-driven programming is essential for creating interactive and responsive applications. Unlike traditional procedural programming, it relies on events—such as user actions or system changes—to control program flow. This lab introduces the concept using C# and Windows Forms in Visual Studio by developing a console application that triggers a custom event when the system time matches user input. This is then extended to a Windows Forms application, where user input is taken through a graphical interface, and visual feedback is provided using form color changes and message boxes. The lab emphasizes practical understanding of event handling, UI controls, and the event-driven model in C#.

Objectives

- To understand the concept of event-driven programming in C#.
- To implement a custom event and event handler using the publisher-subscriber model.
- To design interactive applications using Windows Forms.
- To observe control flow in response to user actions and system events.
- To utilize the Visual Studio Toolbox effectively in Windows Forms development.

Environment Setup

- **Operating System:** Windows 11
- **Software:-**
 1. Visual Studio 2022 (Community Edition)
 2. .NET SDK (latest stable version)
- **Programming Language:** C#
- **Framework:** Windows Forms (.NET Framework)

Methodology and Execution:

Part 1: Console Application for Alarm

This program accepts a user-defined time in *HH:MM:SS* format and continuously checks the system time against it. The *AlarmClock* class defines a custom event, *raiseAlarm* using the delegate *AlarmEventHandler*, and the method *Ring_alarm()* is registered as the event handler. Inside the *Start()* method, a loop compares the system time with the target time each second using *Thread.Sleep(1000)*. When the times match, the event is triggered, and *Ring_alarm()* is called to display a message. Input and output are handled entirely through the console, following the event-driven and publisher-subscriber model.


```

using System;
using System.Threading;

2 references
class AlarmClock
{
    public delegate void AlarmEventHandler(); // Define a delegate
    public event AlarmEventHandler raiseAlarm; // Declare the event

    1 reference
    public void Start(DateTime targetTime)
    {
        while (true)
        {
            DateTime currentTime = DateTime.Now;
            if (currentTime.Hour == targetTime.Hour &&
                currentTime.Minute == targetTime.Minute &&
                currentTime.Second == targetTime.Second)
            {
                raiseAlarm?.Invoke(); // Raise the event
                break;
            }
            Thread.Sleep(1000); // Check every second
        }
    }
}

```

```

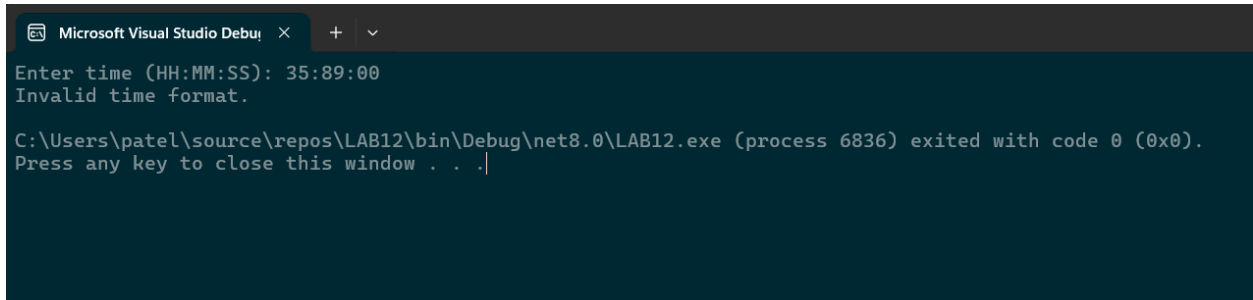
1 reference
public void Ring_alarm()
{
    Console.WriteLine("ALARM! Time matched: " + DateTime.Now.ToLongTimeString());
}

0 references
static void Main(string[] args)
{
    Console.Write("Enter time (HH:MM:SS): ");
    string input = Console.ReadLine();

    if (DateTime.TryParseExact(input, "HH:mm:ss", null, System.Globalization.DateTimeStyles.None, out DateTime userTime))
    {
        AlarmClock alarm = new AlarmClock();
        alarm.raiseAlarm += alarm.Ring_alarm; // Subscribe to the event
        Console.WriteLine("Alarm set for " + userTime.ToLongTimeString());
        alarm.Start(userTime);
    }
    else
    {
        Console.WriteLine("Invalid time format.");
    }
}

```

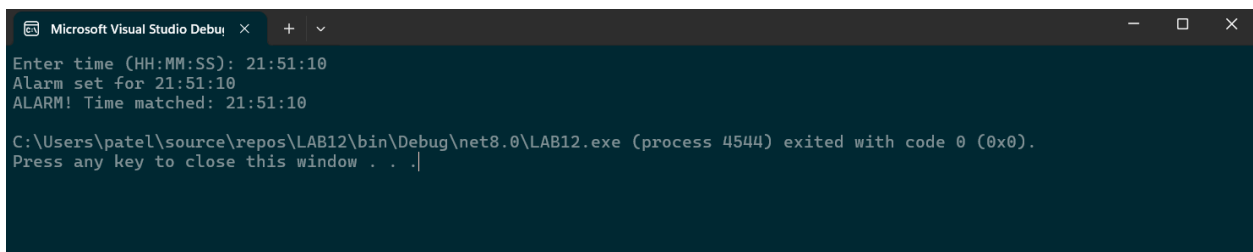
Code Implementation for console applications



A screenshot of the Microsoft Visual Studio Debug Console window. The window has a dark background and a title bar with the text "Microsoft Visual Studio Debug" and a close button. The console output shows the following text: "Enter time (HH:MM:SS): 35:89:00", "Invalid time format.", "C:\Users\patel\source\repos\LAB12\bin\Debug\net8.0\LAB12.exe (process 6836) exited with code 0 (0x0).", and "Press any key to close this window . . .|".

```
Microsoft Visual Studio Debug × + v
Enter time (HH:MM:SS): 35:89:00
Invalid time format.
C:\Users\patel\source\repos\LAB12\bin\Debug\net8.0\LAB12.exe (process 6836) exited with code 0 (0x0).
Press any key to close this window . . .|
```

Output on invalid time



A screenshot of the Microsoft Visual Studio Debug Console window. The window has a dark background and a title bar with the text "Microsoft Visual Studio Debug" and a close button. The console output shows the following text: "Enter time (HH:MM:SS): 21:51:10", "Alarm set for 21:51:10", "ALARM! Time matched: 21:51:10", "C:\Users\patel\source\repos\LAB12\bin\Debug\net8.0\LAB12.exe (process 4544) exited with code 0 (0x0).", and "Press any key to close this window . . .|".

```
Microsoft Visual Studio Debug × + v
Enter time (HH:MM:SS): 21:51:10
Alarm set for 21:51:10
ALARM! Time matched: 21:51:10
C:\Users\patel\source\repos\LAB12\bin\Debug\net8.0\LAB12.exe (process 4544) exited with code 0 (0x0).
Press any key to close this window . . .|
```

Output on valid time and also Alarm gets print at target time

Part 2: Windows Forms App for Event-Driven Alarm

Form1.cs

- Handles the core logic of the application.
- Parses user input for the target time in HH:MM:SS format.
- Starts a timer to change the form's background color every second.
- Stops the color change when the target time is reached.
- Displays a message box notifying the user upon completion.

Form1.Designer.cs

- Defines and initializes the user interface elements of the form.
- Includes a label, textbox for time input, and a start button.
- Sets up the timer control and its properties.
- Configures form size, layout, and startup position.
- Wires up event handlers for UI interaction.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace TimeColorChanger
{
    0 references
    internal static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        0 references
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Program.cs

```

// Form1.cs
using System;
using System.Drawing;
using System.Windows.Forms;

namespace TimeColorChanger
{
    3 references
    public partial class Form1 : Form
    {
        private DateTime targetTime;
        private Random random = new Random();

        1 reference
        public Form1()
        {
            InitializeComponent();
            timer1.Interval = 1000; // 1 second
            timer1.Tick += Timer1_Tick;
        }

        1 reference
        private void btnStart_Click(object sender, EventArgs e)
        {
            if (TimeSpan.TryParse(txtTime.Text.Trim(), out TimeSpan ts))
            {
                targetTime = DateTime.Today.Add(ts);

                // If target is already past for today, assume tomorrow
                if (targetTime <= DateTime.Now)
                {
                    targetTime = targetTime.AddDays(1);
                }

                btnStart.Enabled = false;
                txtTime.Enabled = false;
                timer1.Start();
            }
            else
            {
                MessageBox.Show("Please enter a valid time in HH:MM:SS format.", "Invalid Input", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                txtTime.Focus();
            }
        }

        1 reference
        private void Timer1_Tick(object sender, EventArgs e)
        {
            // Change background to a random color
            this.BackColor = Color.FromArgb(
                random.Next(256),
                random.Next(256),
                random.Next(256));

            // Check if we've reached or passed the target time
            if (DateTime.Now >= targetTime)
            {
                timer1.Stop();
                MessageBox.Show($"Target time reached: {targetTime:HH:mm:ss}", "Done", MessageBoxButtons.OK, MessageBoxIcon.Information);
                btnStart.Enabled = true;
                txtTime.Enabled = true;
            }
        }
    }
}

```

Form1.cs

```

// Form1.Designer.cs
namespace TimeColorChanger
{
    3 references
    partial class Form1
    {
        private System.ComponentModel.IContainer components = null;

        0 references
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        1 reference
        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.txtTime = new System.Windows.Forms.TextBox();
            this.btnStart = new System.Windows.Forms.Button();
            this.timer1 = new System.Windows.Forms.Timer(this.components);
            this.lblPrompt = new System.Windows.Forms.Label();
            this.SuspendLayout();

            this.lblPrompt.AutoSize = true;
            this.lblPrompt.Location = new System.Drawing.Point(12, 20);
            this.lblPrompt.Name = "lblPrompt";
            this.lblPrompt.Size = new System.Drawing.Size(136, 13);
            this.lblPrompt.TabIndex = 0;
            this.lblPrompt.Text = "Enter time (HH:MM:SS) here:";
        }
    }
}

```

```

        this.txtTime.Location = new System.Drawing.Point(154, 17);
        this.txtTime.Name = "txtTime";
        this.txtTime.Size = new System.Drawing.Size(100, 20);
        this.txtTime.TabIndex = 1;

        this.btnStart.Location = new System.Drawing.Point(270, 15);
        this.btnStart.Name = "btnStart";
        this.btnStart.Size = new System.Drawing.Size(75, 23);
        this.btnStart.TabIndex = 2;
        this.btnStart.Text = "Start";
        this.btnStart.UseVisualStyleBackColor = true;
        this.btnStart.Click += new System.EventHandler(this.btnStart_Click);

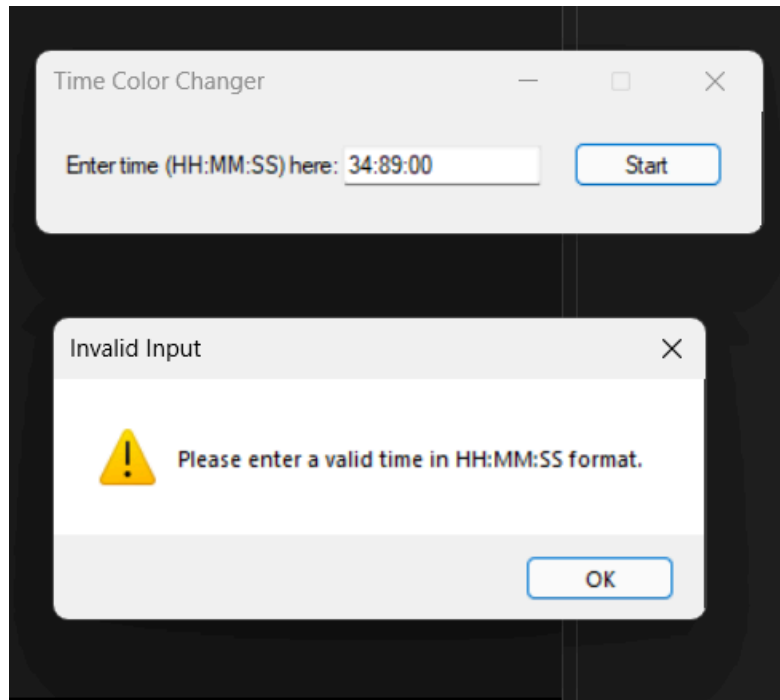
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(364, 61);
        this.Controls.Add(this.btnStart);
        this.Controls.Add(this.txtTime);
        this.Controls.Add(this.lblPrompt);
        this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedDialog;
        this.MaximizeBox = false;
        this.Name = "Form1";
        this.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen;
        this.Text = "Time Color Changer";
        this.ResumeLayout(false);
        this.PerformLayout();
    }

    #endregion

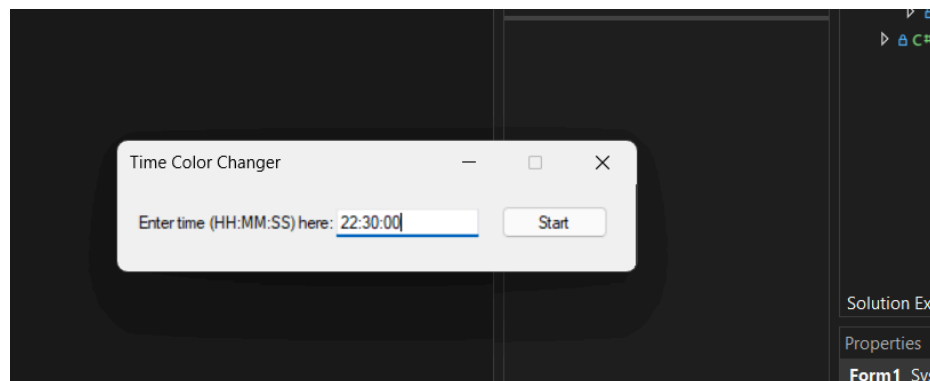
    private System.Windows.Forms.TextBox txtTime;
    private System.Windows.Forms.Button btnStart;
    private System.Windows.Forms.Timer timer1;
    private System.Windows.Forms.Label lblPrompt;
}

```

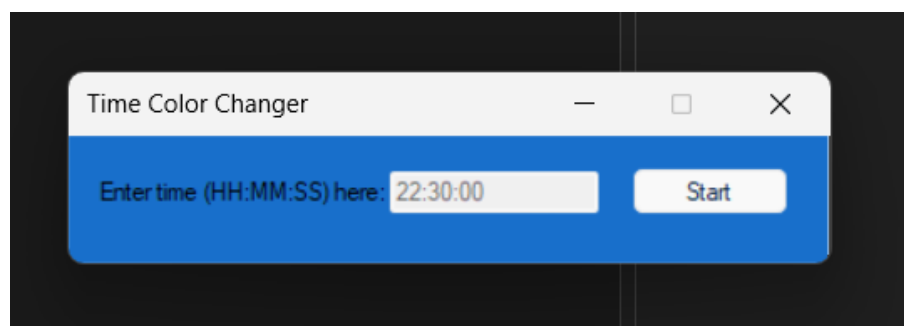
Form1.Designer.cs



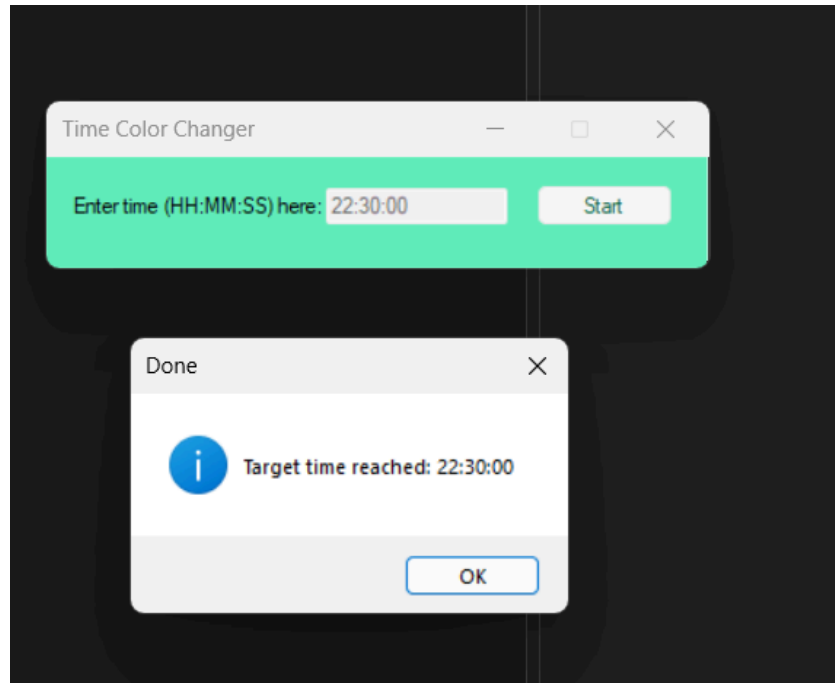
Output on Invalid time



Entered target time



Changing color



Target time achieved

Results and Analysis:

Part 1: Console Application for Alarm

- The console-based alarm application effectively takes a target time from the user and keeps checking the current system time.
- When the system time aligns with the specified input, it alerts the user by displaying ALARM: Time matched on the console.

Part 2: Windows Forms App for Event-Driven Alarm

- The application successfully changes the form's background color every second.
- When the entered time is reached, the timer stops automatically.
- A message box pops up to notify the user that the target time has been reached.
- The Windows Form UI makes the application interactive and eliminates the need for a console.

Challenges

- Implementing a user-defined event using the publisher/subscriber model took effort to structure correctly.
- Ensuring real-time clock checks without freezing the UI or console required understanding of timers or threads.
- Designing the Windows Form UI while keeping it responsive was tricky.
- Avoiding console usage completely in the Windows Forms version needed careful UI planning.

Learning

- Understood how to work with time data in C# using DateTime.
- Learned the concept of custom events and how to implement publisher-subscriber architecture.
- Gained experience with Windows Forms controls like TextBox, Button, and Timer.
- Explored event-driven programming in a graphical interface context.
- Learned how to integrate timers with UI updates in a non-blocking way.

Conclusion

- The lab demonstrated both console-based and GUI-based event handling in C#.
- It highlighted the differences between synchronous console apps and asynchronous GUI apps.
- Implementing events helped reinforce the use of delegates and event handlers.
- The task improved understanding of C# application design in both text and graphical environments.

Overall, it was a valuable exercise in user interaction, time-based events, and UI responsiveness.